# Computing Grounded Extensions of Abstract Argumentation Frameworks

Samer Nofal[a], Katie Atkinson[b], Paul E. Dunne[b]

[a] *Department of Computer Science, German Jordanian University, Jordan*
[b] *Department of Computer Science, University of Liverpool, United Kingdom*

**Abstract**

An *abstract argumentation framework* is a directed graph $(V, E)$ such that the vertices of $V$ denote *abstract arguments* and $E \subseteq V \times V$ represents the *attack* relation between them. We present a new *ad hoc* algorithm for computing the *grounded extension* of an abstract argumentation framework. We show that the new algorithm runs in $\mathcal{O}(|V| + |E|)$ time. In contrast, the existing state-of-the-art algorithm runs in $\mathcal{O}(|V| + |S||E|)$ time where $S$ is the grounded extension of the input graph.

*Keywords:* directed graph, algorithm, grounded extension, argumentation graph, argumentation semantics, abstract argumentation framework

## 1. Introduction

The role of *grounded extensions* (to be defined shortly) has been widely studied in the context of *abstract argumentation frameworks* (AFs), which were originally introduced in Dung's seminal paper [1] and since then have been researched extensively, see for example [2, 3, 4, 5, 6]. AFs are basically directed graphs with vertices representing *abstract arguments* and arrows denoting *attack* relations between them. AFs have been proven useful for modeling decision-support systems in different application areas such as agriculture, e-government, medical care, and legal services, see for example [7, 8, 9, 10].

We now give the definition of grounded extensions.

**Definition 1** (Grounded Extensions)**.** *Let $G = (V, E)$ be a directed graph with $V$ being a set of vertices and $E \subseteq V \times V$ being a binary relation on $V$, and let $F : 2^V \to 2^V$ be a mapping defined as*

$$F(V') = \{x \mid \text{ for every } (y, x) \in E \text{ there is } z \in V' \text{ with } (z, y) \in E\},$$

*then $S \subseteq V$ is the grounded extension of $G$ if and only if*

$$S = \bigcup_{i=1}^{\infty} F^i(\emptyset).$$

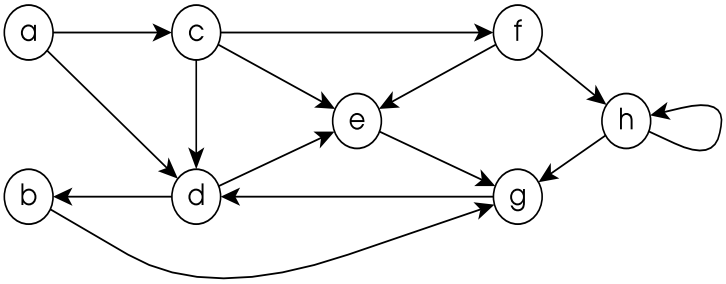


Figure 1: A directed graph $G_1$.

In fact, Dung [1] defines the grounded extension by the least fixpoint of the function $F$. Dung's definition is equivalent to Definition 1 for finitary graphs, including finite graphs which are the focus of this paper.

For a given directed graph $G = (V, E)$, we say that $x$ is a *predecessor* of $y$, and $y$ is a *successor* of $x$ whenever $(x, y) \in E$. For a set $S \subseteq V$, we denote by $S^-$ (respectively $S^+$) the set of all predecessors of the vertices of $S$ (respectively the set of all successors of the vertices of $S$). Further, we say $x$ is *grounded* in $G$ if and only if $x$ is in the grounded extension of $G$.

In this paper we are concerned with the problem of computing the grounded extension of a given directed graph. Take the directed graph $G_1$ of Figure 1. Then, the grounded extension of $G_1$ is computed as follows: $F^1(\emptyset) = \{a\}$, $F^2(\emptyset) = \{a, b, f\}$, $F^3(\emptyset) = \{a, b, f\}$, now we note that $F^3(\emptyset) = F^2(\emptyset)$, and so $\{a, b, f\}$ is the grounded extension of $G_1$.

To the best of our knowledge, the first attempt to formalize an *ad hoc* algorithm for computing grounded extensions was by Modgil and Caminada [11]. Since then, several works in the literature have been devoted to the implementation of the algorithm of Modgil and Caminada, namely those of Nofal et al [12], Gordon [13], Geilen and Thimm [14] and Rodrigues [15]. In fact, all these implementations have a similar core structure as they all are directly based on the algorithm of Modgil and Caminada. Another line of research considers computing grounded extensions by reduction-based methods where the problem instance at hand is reduced to another form and then solved by an off-the-shelf system, for further information on reduction-based systems see for example [16, 17].

In this paper we show that for a given directed graph $G = (V, E)$, with $S$ being its grounded extension, the state-of-the-art implementation of the algorithm of Modgil and Caminada computes $S$ in $\mathcal{O}(|V| + |S||E|)$ time. Then, we propose an enhanced implementation that runs in $\mathcal{O}(|V| + |E|)$ time.

The rest of the paper is organized as follows. In section 2 we recall the state-of-the-art *ad hoc* implementation for computing the grounded extension. Then,

**Algorithm 1:** The algorithm of Modgil and Caminada [11]

**input :** a directed graph $G = (V, E)$.
**output:** $S \subseteq V$ the grounded extension of $G$.

1 $I_0 \leftarrow \emptyset$;
2 $O_0 \leftarrow \emptyset$;
3 **repeat**
4 $\quad I_{i+1} \leftarrow I_i \cup \{x \mid x \notin I_i \cup O_i,\ and\ \forall y :\ if\ (y, x) \in E\ then\ y \in O_i\}$;
5 $\quad O_{i+1} \leftarrow O_i \cup \{x \mid x \notin I_i \cup O_i,\ and\ \exists y : (y, x) \in E\ with\ y \in I_{i+1}\}$;
6 **until** $I_i = I_{i+1}$;
7 $S \leftarrow I_i$;

in section 3 we introduce our new algorithm in two different styles: recursive and non-recursive; then, we give our time complexity results for the new algorithm in contrast to the state of the art. In section 4 we verify experimentally the efficiency of the new algorithm. We conclude the paper in section 5.

## 2. The State-of-the-art Algorithm

Algorithm 1 represents the algorithm of Modgil and Caminada [11]. The algorithm is somewhat a direct formulation from the definition of grounded extensions. However, Algorithm 1 uses two dynamic sets denoted by $I$ and $O$. The $I$ set includes grounded vertices and the $O$ set holds the successors of the vertices currently contained in $I$.

Note that Algorithm 1 was given in [11] at a high level of abstraction such that several aspects of the underlying computations are left unspecified. Before we discuss the state-of-the-art implementation of the algorithm of Modgil and Caminada we show how Algorithm 1 works. In computing the grounded extension of $G_1$ (see Figure 1), the algorithm goes through the following states:

$$\begin{array}{ll} I_0 = \emptyset & O_0 = \emptyset \\ I_1 = \{a\} & O_1 = \{c, d\} \\ I_2 = \{a, b, f\} & O_2 = \{c, d, e, g, h\} \\ I_3 = \{a, b, f\} & O_3 = \{c, d, e, g, h\} \end{array}$$

As $I_2 = I_3$, we conclude with $\{a, b, f\}$ being the grounded extension of $G_1$.

As we noted earlier, a number of computational aspects of Algorithm 1 are open for further developments. Therefore, our aim in this paper is to set efficient actions for the following issues of Algorithm 1:

ISSUE 1 the issue of checking whether a vertex is in $I \cup O$.
ISSUE 2 the issue of deciding if all the predecessors of some vertex are in $O$.
ISSUE 3 the issue of finding those vertices that have a predecessor in $I$.
ISSUE 4 the issue of identifying those vertices that might enter $I$.

**Algorithm 2:** The algorithm of Nofal et al [12].

**input :** a directed graph $G = (V, E)$.
**output:** $S \subseteq V$ the grounded extension of $G$.

```
1 label : V → {in, out, undecided};
2 foreach x ∈ V do
3     label(x) ← undecided;
4 while ∃x with label(x) = undecided s.t. ∀y ∈ {x}⁻ label(y) = out do
5     label(x) ← in;
6     foreach z ∈ {x}⁺ do
7         label(z) ← out;
8 S ← {x | label(x) = in};
```

We note that performing the above listed tasks straightforwardly will lead to a slow algorithm. In [11], Modgil and Caminada mentioned that their algorithm can be improved in a number of ways and they gave a thoughtful note on how to make their algorithm faster. More specifically, Modgil and Caminada suggested in [11] a solution to address issue 2 from the above list. However, most likely their observation was never taken a step further to the point of a fully specified implementation. We elaborate on this in the next section.

In [12], Nofal et al presented a full implementation of the algorithm of Modgil and Caminada. As said earlier, all the *ad hoc* implementations in the literature follow a comparable structure to the one presented by Nofal et al [12], which we recall in Algorithm 2. Later in this section we discuss how Algorithm 2 resolves 1 & 3 from the issues listed earlier.

Having a closer look at Algorithm 2, the algorithm employs a total function $label : V \rightarrow \{in, out, undecided\}$ that maps every vertex in a given directed graph $G = (V, E)$ to a label in $\{in, out, undecided\}$. Initially all vertices of $G$ are mapped to *undecided*. Subsequently, a vertex $x$ with $label(x) = undecided$ will be re-mapped to *in* if and only if

$$\forall y \in \{x\}^{-}\ label(y) = out.$$

On the other hand, a vertex $x$ with $label(x) = undecided$ will be re-mapped to *out* if and only if

$$\exists y \in \{x\}^{-}\ with\ label(y) = in.$$

This re-mapping process goes on until there is no vertex left satisfying the condition under which a vertex is labelled with *in*. At this point, the set of vertices, that are mapped to *in*, forms the grounded extension of the given directed graph. Figure 2 shows an execution of Algorithm 2.

We note that Algorithm 2 failed to address issues 2 & 4. Referring to line 4 in Algorithm 2, note that the algorithm has to scan vertices searching for an *undecided* vertex that might enter the under-construction grounded extension. Additionally, Algorithm 2 checks all predecessors of a given vertex to decide if they are outside the current under-construction grounded extension, again see

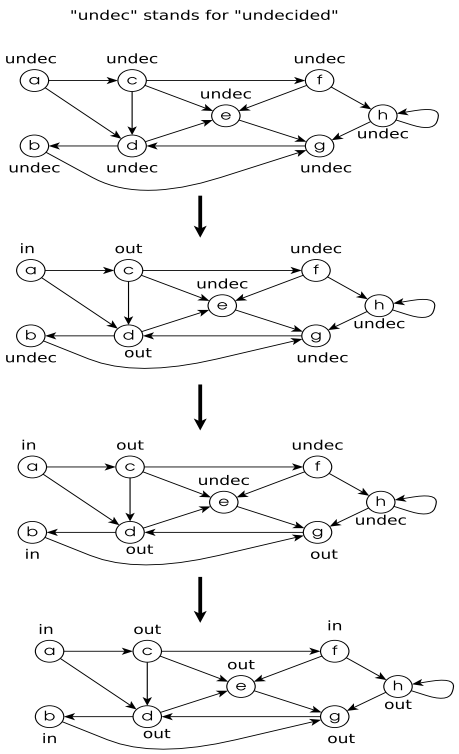


Figure 2: Computing the grounded extension of $G_1$ using Algorithm 2.

line 4 of Algorithm 2. However, Algorithm 2 succeeded in addressing issues 1 & 3. For issue 1, note that if a vertex is labelled with *undecided* then it means that the vertex is not yet in (neither out of) the under-construction grounded extension, and so one simply might check the label of a given vertex to see whether it is currently inside (or outside) the under-construction grounded extension. For issue 3, note that whenever Algorithm 2 maps a vertex, say $x$, to *in*, the algorithm maps the successors of $x$ to *out* and so the algorithm eliminates completely the necessity of searching for a vertex that has a predecessor mapped to *in*.

In the next section we present a new, more efficient algorithm for computing grounded extensions, taking into account all the listed issues (1-4).

## 3. The New Algorithm

We introduce a new, improved algorithm for computing the grounded extension of a given directed graph. We give recursive and non-recursive specifications for the new algorithm. Later in this section we show how the new algorithm addressed successfully the earlier four computational issues of building grounded

extensions. To this end, we start by defining some helpful constructs.

To address the concerns raised in issue 2 & 4, we utilize two structures: *und_pre* and *to_be_in*. The construct *und_pre* keeps track of the number of *undecided* predecessors of a given vertex. Note that the use of *und_pre* is inspired by the remark of Modgil and Caminada [11] that hints at an enhancement for solving issue 2.

**Definition 2** (undecided predecessors)**.** *Let $G = (V, E)$ be a directed graph, $label : V \to \{in, out, undecided\}$ be a total mapping, $\mathbb{N}_0$ be the set of non-negative integers and $und\_pre : V \to \mathbb{N}_0$ be a mapping, then for every $x \in V$ with $label(x) = undecided$ it holds that*

$$und\_pre(x) = |\{y : y \in \{x\}^- \text{ with } label(y) = undecided\}|.$$

The advantage of using *und_pre* is that every time a vertex, say $x$, is mapped to *out* then for every $y \in \{x\}^+$ with $label(y) = undecided$ we update $und\_pre(y) \leftarrow und\_pre(y) - 1$. Thus, if $und\_pre(y)$ becomes zero, then we conclude that $y$ is grounded. Note that by using *und_pre* we address issue 2 that raises the concern about finding a fast way to check, for a given vertex, whether all of its predecessors are outside the current under-construction grounded extension.

We turn to another useful construct that we call *to_be_in*, which is basically a set holding those vertices that are grounded but not yet included in the under-construction grounded extension. We define the condition under which a vertex enters the set *to_be_in*.

**Definition 3** (*to_be_in* vertices)**.** *Let $G = (V, E)$ be a directed graph and $label : V \to \{in, out, undecided\}$ be a total mapping, then the set $to\_be\_in \subseteq V$ is defined by*

$$to\_be\_in = \{x \mid label(x) = undecided \text{ with } und\_pre(x) = 0\}.$$

Now we explain the computational benefit of *to_be_in*. Note that if a vertex, say $x$, joins an under-construction grounded extension then $x$ will expel all of its successors, and consequently some vertices may become grounded. This process repeatedly continues until there are no more vertices that need to join the under-construction grounded extension. To control the scope of those vertices that are affected by a change in the under-construction grounded extension, we use the structure *to_be_in* to hold such vertices, which truly need to be processed further, and thus we avoid checking vertices in the graph over and over. Consequently, we resolved issue 4 by using the set *to_be_in*.

We now give Algorithm 3 that builds the grounded extension of a given directed graph. Figure 3 shows a run of Algorithm 3. Although the figure shows the same number of states as shown in the behavior of the state-of-the-art algorithm in Figure 2, we note that the computational benefit of the additional structures (i.e. *to_be_in* and *und_pre*) is evident in computing the next set of vertices that need to be included in the under-construction grounded extension.

**Algorithm 3:** (new) non-recursive computing of grounded extensions.

**input :** a directed graph $G = (V, E)$.
**output:** $S \subseteq V$ the grounded extension of $G$.

```
1  label : V → {in, out, undecided};
2  und_pre : V → ℕ0;
3  to_be_in ← ∅;
4  foreach x ∈ V do
5      label(x) ← undecided ;
6      und_pre(x) ← |{x}⁻|;
7      if und_pre(x) = 0 then
8          to_be_in ← to_be_in ∪ {x};
9  while to_be_in ≠ ∅ do
10     remove x from to_be_in;
11     label(x) ← in;
12     foreach y ∈ {x}⁺ with label(y) ≠ out do
13         label(y) ← out;
14         foreach z ∈ {y}⁺ with label(z) = undecided do
15             und_pre(z) ← und_pre(z) − 1;
16             if und_pre(z) = 0 then
17                 to_be_in ← to_be_in ∪ {z};
18 S ← {x | label(x) = in};
```

Observe, without using these new structures we need to scan all vertices to identify a new grounded vertex, see line 4 of Algorithm 2. In contrast, with using *to_be_in* and *und_pre* we are able to find more efficiently those vertices that must join the under-construction grounded extension, see line 9 of Algorithm 3. In other words, in each iteration of the loop at line 4 of Algorithm 2 we search in the whole set of vertices $V$ to find a grounded vertex, whereas in Algorithm 3 we do not search for grounded vertices because they are ready to pick from the set *to_be_in*, again see line 9 of Algorithm 3.

We now present theorem 1 and 2 respectively to argue about the correctness of algorithm 3 and its time complexity.

**Theorem 1.** *Let $G = (V, E)$ be a directed graph. Algorithm 3 computes $S$, the grounded extension of $G$.*

*Proof.* According to Definition 1, we need to show that $S = \cup_{i=1}^{\infty} F^i(\emptyset)$. For this, we first prove inductively that $\cup_{i=1}^{\infty} F^i(\emptyset) \subseteq S$. For showing $F^1(\emptyset) \subseteq S$, we note that $F^1(\emptyset) = \{x | \{x\}^- = \emptyset\}$. By the end of executing the *for* loop of the algorithm, *to_be_in* $= \{x | \{x\}^- = \emptyset\}$. Due to lines 10, 11, and 18 of the algorithm $S \supseteq$ *to_be_in*. Therefore, $F^1(\emptyset) \subseteq S$. We now prove that for every positive integer $i$ it holds that $F^i(\emptyset) \subseteq S \implies F^{i+1}(\emptyset) \subseteq S$. Assume $F^i(\emptyset) \subseteq S$. By Definition 1,

$$F^{i+1}(\emptyset) = \{x | \forall y \in \{x\}^- \ \exists z \in \{y\}^- \text{ such that } z \in F^i(\emptyset)\}. \tag{1}$$

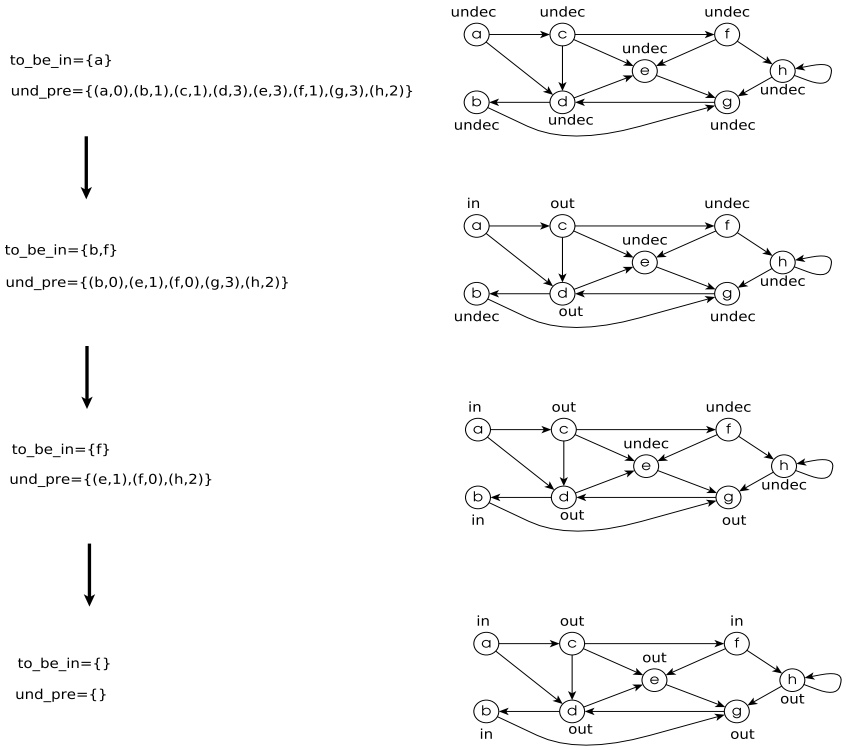


Figure 3: Computing the grounded extension of $G_1$ using Algorithm 3.

As $F^i(\emptyset) \subseteq S$, and $F$ is a monotonic function, it follows that $F(F^i(\emptyset)) \subseteq F(S)$. Hence,

$$F^{i+1}(\emptyset) \subseteq \{x | \forall y \in \{x\}^- \ \exists z \in \{y\}^- \text{ such that } z \in S\}. \tag{2}$$

At the end of the algorithm it holds that

$$\begin{aligned} S &= \{x | und\_pre(x) = 0\} \text{ (see lines 7, 8, 16, and 17)} \\ &= \{x | \forall y \in \{x\}^- \ label(y) = out\} \text{ (see lines 13, 14 and 15)} \\ &= \{x | \forall y \in \{x\}^- \ \exists z \in \{y\}^- \text{ with } label(z) = in\} \text{ (see lines 11-13)} \\ &= \{x | \forall y \in \{x\}^- \ \exists z \in \{y\}^- \text{ such that } z \in S\} \text{ (see line 18)} \end{aligned} \tag{3}$$

Using (3), we rewrite (2) as

$$F^{i+1}(\emptyset) \subseteq S. \tag{4}$$

Therefore, we establish $F^i(\emptyset) \subseteq S \implies F^{i+1}(\emptyset) \subseteq S$.

For proving $S \subseteq \cup_{i=1}^{\infty} F^i(\emptyset)$, we observe that

$$S = \bigcup_{k=1}^{n} to\_be\_in^k. \tag{5}$$

For $k = 1$, $to\_be\_in^k$ denotes the state of $to\_be\_in$ just at the end of executing the *for* loop. For $k > 1$, $to\_be\_in^k$ designates the state of $to\_be\_in$ at the end of executing round $k-1$ of the *while* loop. Thus, we now prove that $to\_be\_in^1 \subseteq \cup_{i=1}^{\infty} F^i(\emptyset)$. Note that by the end of executing the *for* loop, $to\_be\_in^1 = \{x | \{x\}^- = \emptyset\}$, see lines 6-8. As $F^1(\emptyset) = \{x | \{x\}^- = \emptyset\}$, $to\_be\_in^1 \subseteq \cup_{i=1}^{\infty} F^i(\emptyset)$ is established. Now we need to prove that for every positive integer $k$

$$\cup_{j=1}^{k} to\_be\_in^j \subseteq \cup_{i=1}^{\infty} F^i(\emptyset) \implies \cup_{j=1}^{k+1} to\_be\_in^j \subseteq \cup_{i=1}^{\infty} F^i(\emptyset). \tag{6}$$

Suppose $\cup_{j=1}^{k} to\_be\_in^j \subseteq \cup_{i=1}^{\infty} F^i(\emptyset)$. Now, we note that

$$\cup_{j=1}^{k+1} to\_be\_in^j = \{x | und\_pre(x) = 0\}^{k+1}, \tag{7}$$

where $\{x | und\_pre(x) = 0\}^{k+1}$ denotes the state by the end of executing round $k$ of the *while* loop, see lines 16 and 17. Due to lines 13-15, we rewrite (7) as

$$\cup_{j=1}^{k+1} to\_be\_in^j = \{x | \forall y \in \{x\}^- label(y) = out\}^{k+1}. \tag{8}$$

Referring to lines 11-13 in the algorithm, we rewrite (8) as

$$\cup_{j=1}^{k+1} to\_be\_in^j = \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ with } label(z) = in\}^{k+1}. \tag{9}$$

Considering lines 10-11 in the algorithm, we rewrite (9) as

$$\cup_{j=1}^{k+1} to\_be\_in^j = \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in \cup_{j=1}^{k} to\_be\_in^j\}. \tag{10}$$

As $\cup_{j=1}^{k} to\_be\_in^j \subseteq \cup_{i=1}^{\infty} F^i(\emptyset)$,

$$\cup_{j=1}^{k+1} to\_be\_in^j \subseteq \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in \cup_{i=1}^{\infty} F^i(\emptyset)\}. \tag{11}$$

Recall,

$$\begin{aligned} \cup_{i=1}^{\infty} F^i(\emptyset) &= F^1(\emptyset) \cup F^2(\emptyset) \cup F^3(\emptyset) \cup ... \\ &= \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in \emptyset\} \cup \\ &\quad \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in F^1(\emptyset)\} \cup \\ &\quad \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in F^2(\emptyset)\} \cup ... \\ &= \{x | \forall y \in \{x\}^- \; \exists z \in \{y\}^- \text{ such that } z \in \cup_{i=1}^{\infty} F^i(\emptyset)\}. \end{aligned} \tag{12}$$

Using (12), we may rewrite (11) as

$$\cup_{j=1}^{k+1} to\_be\_in^j \subseteq \cup_{i=1}^{\infty} F^i(\emptyset), \tag{13}$$

which completes our proof. □

**Theorem 2.** *For a given directed graph $G = (V, E)$, Algorithm 3 computes $S$, the grounded extension of $G$, in $\mathcal{O}(|V| + |E|)$ time.*

*Proof.* Evidently, the loop at line 4 of the algorithm iterates $|V|$ times. Also, lines 10 and 11 are executed at most $|V|$ times, while the remaining part of the loop runs in $\mathcal{O}(|E|)$ time. This is because:

- for every $x \in to\_be\_in$, the inner loop at line 12 runs $|\{x\}^+|$ times and,
- for every $y \in \{x\}^+$ with $label(y) \neq out$, the inner loop at line 14 runs $|\{y\}^+|$ times,

Hence, the loop (starting at line 9) examines every edge in $E$ at most once, and so the loop runs in $\mathcal{O}(|V| + |E|)$ time. □

Now we analyze the time efficiency of Algorithm 2.

**Theorem 3.** *For a given directed graph $G = (V, E)$, Algorithm 2 computes $S$, the grounded extension of $G$, in $\mathcal{O}(|V| + |S||E|)$ time.*

*Proof.* The loop at line 2 of the algorithm iterates $|V|$ times. Nonetheless, the loop starting at line 4 runs in $\mathcal{O}(|S||E|)$ time because of the following:

- Since the task of the loop is to discover grounded vertices gradually (i.e. one vertex in each round), the loop keeps iterating while there is still an unexplored grounded vertex $x$. As all explored grounded vertices are eventually saved in $S$ (line 8), the loop performs $|S|$ times.
- However, in each round of the loop, evaluating the loop-continuation condition (line 4) runs in the order of $|E|$ since it may require (in a worst-case scenario) exploring the predecessors of every $x \in V$. □

Now we present Algorithm 4, which is a recursive definition that is similar to the essence of Algorithm 3. Note that Algorithm 4 builds on the same ideas of Algorithm 3 except the use of the *to_be_in* set where the recursive algorithm does not iterate over such set but instead recursively invokes a function called *include*. Thereby, Algorithm 4 relies on the hosting runtime environment's stack to keep track of the vertices that are qualified to enter the under-construction grounded extension. With respect to efficiency, both algorithms have a similar time complexity. However, the recursive definition is more elegant because it removes the need for an additional explicit structure, i.e. the set *to_be_in*.

Figure 4 illustrates a recursive computing of the grounded extension of a given graph. In comparing the new algorithms' behavior (see Figure 3 and Figure 4) with the the behavior of Algorithm 2 (see Figure 2), one might wonder what is the speedup gain from the new algorithms given that the number of states has not decreased in contrast to the state-of-the-art algorithm. Responding to that, we stress that the efficiency of the new algorithm lies in reducing significantly the processing time spent in transitioning to a new state from a previous one, as we illustrated throughout this section.

As Algorithm 4 has a similar structure to Algorithm 3 (except having the main loop realized recursively), we omit the correctness proof and the time analysis of Algorithm 4.

**Algorithm 4:** (new) recursive computing of grounded extensions.

**input :** a directed graph $G = (V, E)$.
**output:** $S \subseteq V$ the grounded extension of $G$.

```
1  label : V → {in, out, undecided};
2  und_pre : V → ℕ0;
3  Function include(x)
4      label(x) ← in;
5      foreach y ∈ {x}+ with label(y) ≠ out do
6          label(y) ← out;
7          foreach z ∈ {y}+ with label(z) = undecided do
8              und_pre(z) ← und_pre(z) − 1;
9              if und_pre(z) = 0 then
10                 include(z);
11 Function main()
12     foreach x ∈ V do
13         label(x) ← undecided ;
14         und_pre(x) ← |{x}−|;
15     foreach x with label(x) = undecided s.t. und_pre(x) = 0 do
16         include(x);
17     S ← {x | label(x) = in};
```

Note that the algorithms, presented in this paper, can be easily modified to solve the problem of deciding whether a given vertex is grounded in a given directed graph. Observe, we can terminate the algorithm immediately after the vertex in question has been included in the under-construction grounded extension.

In the next section we experimentally verify the time efficiency of the new algorithms.

## 4. Efficiency Verification

We aim to verify practically that the new algorithm (Algorithm 3) is more efficient than the state-of-the-art algorithm (Algorithm 2). In particular, we compare the new algorithm against three known systems: heureka [14], Carneades [13], and CoQuiAAS [18]. Carneades and heureka implement a procedure similar to Algorithm 2, whereas CoQuiASS is in general a reduction-based solver. Recall, a reduction-based system is a solver that converts a given problem instance into another form that will then be solved by a ready-made system. With respect to computing grounded extensions, heureka and CoQuiAAS are the best systems according to the results of the second international competition on computational models of argumentation 2017 (ICCMA17) [19], whereas Carneades was the best solver according to the results of the first international competition on computational models of argumentation 2015 (ICCMA15) [17].

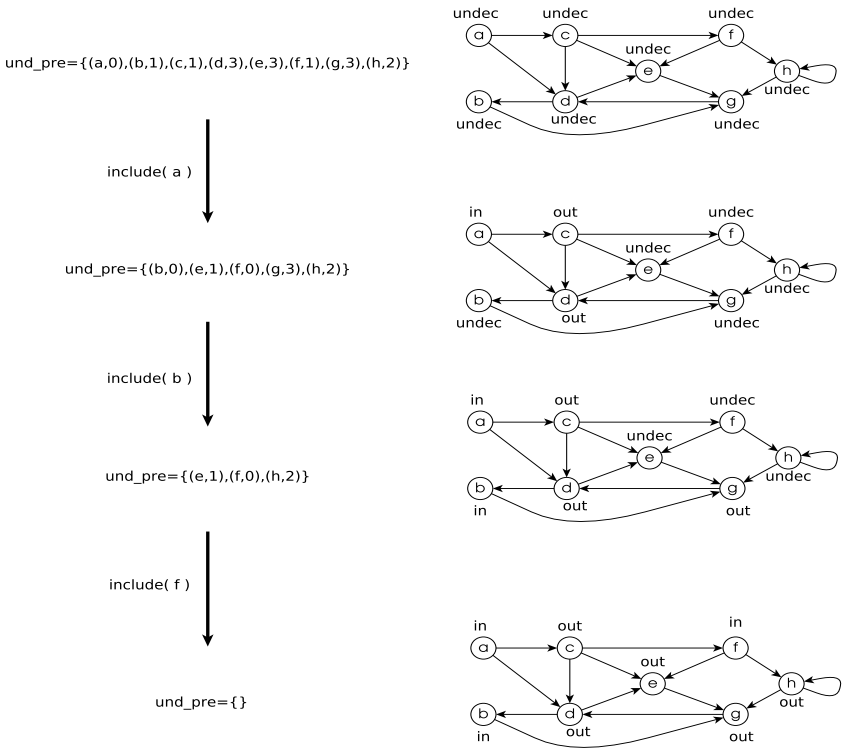


Figure 4: Computing the grounded extension of $G_1$ using Algorithm 4.

We executed Algorithm 3[1] on a machine equipped with an Intel-Core-i7 processor and 4 gigabytes of system memory. On our machine we carried out the evaluation using *benchmark C* that is adopted by ICCMA17 for evaluating competitors in computing grounded extensions. Benchmark C is a set of 350 directed graphs of different shape and size. For each problem instance we set a timeout of ten minutes and allocated one gigabyte of memory.

Table 1 reports for each listed system the total number of solved problem instances along with the total elapsed time (in seconds). Note that the "total elapsed time" does not include the elapsed time of the trials that timeout.

According to the ICCMA17 standards, if a system is able to solve more problem instances than another can do within a predefined timeout, then it is considered more efficient. In summary, the new algorithm solved three more problem instances and in total time notably shorter than the time reported by the best existing systems.

In fact, the solvers ArgTools [12] and EqArgSolver [15] implement an *ad hoc*

[1]The C++ source code is available at https://sourceforge.net/projects/argtools/.

Table 1: Summary of our results.

| system | total elapsed time | number of solved instances | timeouts |
|---|---|---|---|
| Algorithm 3 | 121 | 350 | 0 |
| CoQuiAAS | 333 | 347 | 3 |
| heureka | 476 | 347 | 3 |
| Carneades | 370 | 343 | 7 |

algorithm similar to Algorithm 2. Nonetheless, we did not include EqArgSolver and ArgTools in this evaluation because heureka and CoQuiAAS seem to be more efficient according to the results of ICCMA17 [19].

Observe, the environment machine of ICCMA17 is equipped with an Intel-Xeon processor alongside 16 gigabytes of system memory; however, only four gigabytes were allocated for each problem instance. According to the results of ICCMA17 [19], heureka and CoQuiAAS have solved only 345 problem instances, whereas our experiment showed that the two systems actually solved 347 problem instances. Particularly, both systems solved two more graphs: admbuster_500000 and admbuster_1000000. As our machine is inferior to the environment of ICCMA17, we conjecture that there might be an error returned by the solvers to the environment of ICCMA17 when they are executed on the two graphs. This especially might be true because ICCMA17 reported that heureka and CoQuiAAS indeed finished execution on the two mentioned graphs within the timeout limit but still received a zero score. According to the ICCMA17 rules, a solver scores a 1, if it delivers the correct result; $-5$, if it delivers an incorrect result; 0 otherwise. Since we observed in our experiments that heureka and CoQuiAAS solved successfully the two aforementioned benchmark graphs, it might be a configuration issue that renders the two solvers inconsistent with the ICCMA17 environment.

## 5. Conclusion

We presented a new, more efficient algorithm for constructing grounded extensions of abstract argumentation frameworks. In addition to the theoretical evaluation we presented, we empirically verified that the new algorithms build grounded extensions faster than three well-known systems. We discussed two different implementations for the new algorithm: recursive and non-recursive. Both implementations have comparable time efficiency but they work in different memory spaces. Recall, recursive definitions run on the environment's stack memory, which might be initially restricted. Hence, the target environment might need to be prepared to provide larger stack memory for the hosted recursive implementation. In contrast, the non-recursive implementation employs a traditional loop that runs with a dynamically-allocated space, which is more flexible and requires no further actions to prepare the target environment. Consequently, we note that although the recursive implementation looks more elegant, the non-recursive version is easily portable.

**Acknowledgment**

We thank the anonymous reviewers for their helpful comments that improved the presentation of this article.

**References**

[1] Dung, P. M. (1995) On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, **77**, 321–358.

[2] Baroni, P., Caminada, M., and Giacomin, M. (2011) An introduction to argumentation semantics. *Knowledge Eng. Review*, **26**, 365–410.

[3] Atkinson, K., Baroni, P., Giacomin, M., Hunter, A., Prakken, H., Reed, C., Simari, G. R., Thimm, M., and Villata, S. (2017) Towards artificial argumentation. *AI Magazine*, **38**, 25–36.

[4] Bench-Capon, T. and Dunne, P. E. (2007) Argumentation in artificial intelligence. *Artificial Intelligence*, **171**, 619 – 641.

[5] Modgil, S. et al. (2013) The added value of argumentation. In Ossowski, S. (ed.), *Agreement Technologies*, Law, Governance and Technology Series, **8**, pp. 357–403. Springer-Verlag, Berlin.

[6] Simari, G. R. and Rahwan, I. (eds.) (2009) *Argumentation in Artificial Intelligence*. Springer-Verlag, Berlin.

[7] Tamani, N., Mosse, P., Croitoru, M., Buche, P., Guillard, V., Guillaume, C., and Gontard, N. (2015) An argumentation system for eco-efficient packaging material selection. *Computers and Electronics in Agriculture*, **113**, 174 – 192.

[8] Hunter, A. and Williams, M. (2012) Aggregating evidence about the positive and negative effects of treatments. *Artificial Intelligence in Medicine*, **56**, 173–190.

[9] Bench-Capon, T. J. M., Atkinson, K., and Wyner, A. Z. (2015) Using argumentation to structure e-participation in policy making. *T. Large-Scale Data- and Knowledge-Centered Systems*, **18**, 1–29.

[10] Bench-Capon, T. J. M. and Prakken, H. (2010) Using argument schemes for hypothetical reasoning in law. *Artif. Intell. Law*, **18**, 153–174.

[11] Modgil, S. and Caminada, M. (2009) Proof theories and algorithms for abstract argumentation frameworks. In Simari and Rahwan [6], pp. 105–129.

[12] Nofal, S., Atkinson, K., and Dunne, P. E. (2014) Algorithms for argumentation semantics: Labeling attacks as a generalization of labeling arguments. *J. Artif. Intell. Res. (JAIR)*, **49**, 635–668.

[13] Gordon, T. F. (2013) Introducing the carneades web application. *International Conference on Artificial Intelligence and Law, ICAIL '13, Rome, Italy, June 10-14, 2013*, pp. 243–244. ACM, New York.

[14] Geilen, N. and Thimm, M. (2017) Heureka: A general heuristic backtracking solver for abstract argumentation. *Theory and Applications of Formal Argumentation - 4th International Workshop, TAFA 2017, Melbourne, VIC, Australia, August 19-20, 2017, Revised Selected Papers*, pp. 143–149. Springer-Verlag, Berlin.

[15] Rodrigues, O. (2017) Eqargsolver - system description. *Theory and Applications of Formal Argumentation - 4th International Workshop, TAFA 2017, Melbourne, VIC, Australia, August 19-20, 2017, Revised Selected Papers*, pp. 150–158. Springer-Verlag, Berlin.

[16] Charwat, G., Dvorák, W., Gaggl, S. A., Wallner, J. P., and Woltran, S. (2015) Methods for solving reasoning problems in abstract argumentation - A survey. *Artif. Intell.*, **220**, 28–63.

[17] Thimm, M. and Villata, S. (2017) The first international competition on computational models of argumentation: Results and analysis. *Artif. Intell.*, **252**, 267–294.

[18] Lagniez, J., Lonca, E., and Mailly, J. (2015) Coquiaas: A constraint-based quick abstract argumentation solver. *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*, pp. 928–935. IEEE, New Jersey.

[19] Gaggl, S. A., Linsbichler, T., Maratea, M., and Woltran, S. (2018) Summary report of the second international competition on computational models of argumentation. *AI Magazine*, **39**, 77–79.